SF298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Reprint

4. TITLE AND SUBTITLE
Title shown on Reprint

5. FUNDING NUMBERS
ARO MIPR 156-94

6. AUTHOR(S)
Author(s) listed on Reprint

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U. S. Army Research Office
P. O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ARO 30989.22-MA

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

ABSTRACT ON REPRINT

19960621 138

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Proceedings of the 1995 Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development: Specification-Based Software Architectures, Monterey, CA, 12-14 Sept 1995

Monterey Workshop '95: Specification-based Software Architectures – Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development

Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

1. Introduction

Current software development capabilities need improvement to effectively produce software that meets users' needs. Formal software models that can be mechanically processed can provide a sound basis for building and integrating tools that produce software faster, cheaper, and more reliably. Formal methods can also increase automation and decrease inconsistency in software development.

The goal of the Monterey Workshop series is to help increase the practical impact of formal methods for software development so that these potential benefits can be realized in actual practice. Each year we focus in depth at one aspect of software development. In 1992, the focus was real-time and concurrent systems; in 1993, software slicing and merging; and in 1994, software evolution. This year's focus is specification-based software architectures.

This workshop helps clarify what good formal methods are and what are their limits. According to Webster's Dictionary, *formal* means definite, orderly, and methodical; it does not necessarily entail logic or proofs or correctness. Everything that computers do is formal in the sense that syntactic structures are manipulated according to definite rules. Formal methods are syntactic in essence and semantic in purpose. Given the motivations of the workshop, we believe this is the most appropriate sense for the word "formal" in the phrase "formal methods." We expect the ultimate main benefits of formal methods to be decision support for and partial automation of the software development process.

2. Scope

Specification-based software architectures address families of software systems with a common problem domain and common solution structures. These domains can be general, and can have many different specializations that are amenable to the same solution structure. The problem domains can have a wide variety, including scientific computing, business information systems, computer-aided design environments, real-time control systems, distributed information systems, and military applications.

We take a specification to be a formal description of the behavior visible at the external interface of a component. A specification typically describes what must happen (liveness constraints), what may not happen (safety constraints), and timing constraints.

A software architecture defines the common structure of a family of systems by specifying the components that comprise systems in the family, the relations and interactions between the components, and the rationale for the design decisions embodied in the structure. The components are subsystems that are used as building blocks in the architecture. They can have a wide variety of scales, and can themselves be defined using lower level software

architectures, resulting in hierarchical descriptions of system structure. The components are encapsulated black boxes. The component slots in an architecture are abstract in the sense that they can be filled by a variety of concrete components that satisfy the requirements of the slots. These requirements are given via the component specifications associated with the architecture.

3. The Significance of Software Architecture

Software architectures are relevant to many aspects of computer-aided software development, including automatic program generation, reuse, evolution, and systems integration.

Connecting components. Any meaningful interaction between subsystems requires a shared conceptual model that can capture the meaning of the interaction. This model serves to enable design using black-box components, because it provides designers a characterization of the behavior of the component that is independent of its realization. The most effective models are abstract in this sense. The models can also only partially constrain the behavior of a component, with the result that a slot in the architecture can be filled with a variety of components that agree on certain aspects of their behavior and differ in others. This enables a single design to provide a controlled spectrum of possible system behaviors.

The conceptual models of component behavior also enable bridging between different concrete realizations of interfaces with the same abstract meaning. Support for software architectures should include automatic means for generating bridging transformations that enable connections between components with common conceptual models but different physical realizations of the same abstract interaction, including differences in data representations and control conventions.

Relation to languages. A software architecture is based on common models of computations. The most useful of these are abstract ones, which can serve to unify a variety of concrete realizations of the same conceptual model.

Practical applications require descriptions of particular architectures. For effective automation support, these descriptions should be expressed in a formal language that embodies the computational models underlying the architecture. Different choices of models lead to different kinds of languages. Overly detailed models of computation can quickly lead to very complicated languages and architecture descriptions. A suitably abstract model is essential for simplicity and practical usefulness.

Relation to program generation. Generating programs from specifications is very difficult, and is probably not tractable in an unconstrained context. A given problem domain and a given set of general solution structures specified by a software architecture can make the problem tractable in practice. In such a situation, the program generator is not required to create new solutions to problems, but only to tailor the general solutions given by the architecture to particular instances of the problem domain addressed by the architecture. Thus the architecture defines the range of problems that can be handled by a given solution generator.

Relation to composition and reuse. One of the difficulties in creating large systems by connecting (composing) smaller systems is compatibility between the conceptual models underlying the subsystems. A software architecture defines a common conceptual model. Components designed or generated to fit the same architecture will have compatible con-

ceptual models, and thus consistent connections can be created, possibly via some bridging code to transform between concrete representation conventions. This approach can prevent severe integration problems that could in the worst case be solvable only by redesigning one or more of the components.

Reuse is subject to a similar difficulty, because independently developed components are unlikely to have completely consistent conceptual models. However, components designed to fit a given architecture can be reused without modification in the scope of that architecture. This is significant because economically effective reuse depends on reuse of components without modification.

Relation to decomposition and analysis. Complex systems are understood by people via hierarchical decomposition. An architecture contains the information about the interface conventions and the requirements associated with the component slots in the architecture that are needed to make this work on a large scale.

Relation to evolution and merging. The constraints and conventions associated with a composite design are essential to determine what can be changed without damaging a design. This is precisely the information recorded in the specification part of a specification-based software architecture. Much of the difficulties with evolution of legacy software stem from the loss of this information.

Software change merging is the process of automatically combining several changes to the same version of a software system. An essential requirement for this process is to detect all potential conflicts between changes, and to guarantee semantic integrity of the results of no conflicts are reported. The behavioral requirement information contained in a software architecture can enhance this process and enable more discriminating results. Recent results show that change merging cannot be done via a divide and conquer approach, which implies that the computational cost of change merging increases faster than linearly with the size of the system. Change merging at the architectural level has been shown to be feasible, and this approach is most promising for large systems because it appears to be computationally tractable.

Relation to networks Networks are the physical means for realizing connections between remote nodes in distributed architectures. Knowledge of the constraints and conventions associated with a connection in a software architecture can in principle be exploited by the network protocols to provide better service.

Relation to hybrid systems Some of the components slots in a software architecture can in principle be filled by components realized in hardware rather than by software. The information in a software architecture can be used to support hardware/software codesign, and can eventually lead to automatic realizations in hardware for some classes of components.

Relation to heterogeneous architectures Different components in a software architecture can in principle be implemented using different programming languages, operating systems, or hardware platforms. A carefully structured description of the software architecture that provides annotations to refine abstract architectures with physical realization attributes can effectively support generation of connections between nodes with different (and hence incompatible) physical realizations, by automatically constructing the required transformations and inserting them in the connections.

4. Workshop Summary

The workshop consists of formal presentations on the subject, interleaved with discussions to bring out implicit assumptions, clarify relationships between different points of view, and make assessments. The conclusions of each session are summarized by the reporters for the session, and the results are integrated into the article "Summary of the '95 Monterey Workshop" on pages 107-112 of this proceedings.